# 序

多年來，我的雇主都很信任我能夠勝任面試的工作。如果我們要找擁有 JavaScript 技能的人，我的第一輪問題會是…好吧，實際上我會先確定候選人是否需要上洗手間或是喝個什麼飲料，因為舒適是很重要的。滿足了候選人的液體攝取或輸出需求後，我會開始了解候選人是否熟悉 JavaScript，或只懂 jQuery。

這並不代表 jQuery 有什麼不好，它能讓你在不是很了解 JavaScript 的情況下做很多事情，而這是一種特色，而非缺點。但如果我們開的職缺需要與 JavaScript 效能和維護有關的進階技能，你就得找到知道像 jQuery 這類程式庫是如何組合而成的高手。你必須能夠駕馭 JavaScript 的核心能力，就跟那些程式庫一樣。

若要大略掌握一個人的 JavaScript 核心技能程度，我最感興趣的會是他們如何運用 closures（你已經讀過了此系列的*你所不知道的 JS ｜範疇與 Closures* 對吧？）以及如何藉由非同步獲取最佳效能，這正是本書的主題。

作者會先帶領初學者了解所謂的 callbacks，也就是非同步程式設計（asynchronous programming）最基本的元素。當然，光是這些基本材料並沒有辦法做出特別令人喜愛的佳餚，不過下一堂課滿滿都是非常美味的 Promises（承諾）！

如果你還不知道 Promises 是什麼，現在正是認識它們的好時機。Promises 是目前在 JavaScript 及 DOM 中提供非同步回傳值（async return values）的正式方法。未來所有的 async DOM API 都會使用它們，而且已經有許多地方用到它們了，所以請做好準備！在本文寫作之時，主要的瀏覽器大多都內建了 Promises 功能，而 IE 也很快就會支援了。

一旦你細細品嘗完 Promises，我希望你的肚子還留有空間準備迎接下一堂課：產生器（Generators）。



碁峰 www.gotop.com.tw

產生器已經偷偷溜進了穩定版的 Chrome 和 Firefox，並沒有引起太多的注意，這是因為，老實說，它們複雜的程度勝過於有趣的程度。又或者說，那是我在看到它們與 Promises 結合運用之前的認知，在那種情況下，它們就變為了提升可讀性與可維護性的重要工具。

至於甜點是什麼，我可不想破壞驚喜，但請做好準備一睹 JavaScript 的未來！這本書涵蓋了能夠讓你以更細緻的方式控制共時（concurrency）與非同步（asynchronicity）的功能。

好啦，不打擾你享受這本書的時間了，直接讓好戲上場吧！如果閱讀本前言之前你已經讀了此書的一部分，那就給自己 10 個非同步點數吧！你應該得到的！

—*Jake Archibald*（*http://jakearchibald.com*、*@jaffathecake*）
*Google Chrome* 的 *Developer Advocate*



碁峰 www.gotop.com.tw

[1]

# 非同步：現在和之後

像 JavaScript 這樣的語言最重要但經常被誤解的部分之一，是如何表達並操作分布在某段時間上的程式行為。

這並不單指從一個 for 迴圈的開始到結束之間發生了什麼事，那當然需要一點時間（微秒到毫秒）來處理。這裡談論的主要是，當你程式的某部分**現在**（*now*）立即執行，而另外一部分要在**之後**（*later*）才執行時，會發生什麼事，在這個 *now* 和 *later* 之間，有個間隙（gap）存在，期間你的程式並沒有積極執行。

幾乎所有曾經被寫出來的實用程式（特別是用 JS 寫的）在某些面向都必須管理這種間隙，無論那發生在等候使用者輸入時、向資料庫或檔案系統請求資料時、透過網路傳送資料並等候回應時，或是固定時間重複執行某項任務（例如製作動畫）的時候。不管是哪種情形，你的程式都必須處理跨越時間間隙的程式狀態，正如同倫敦地鐵著名的「mind the gap」（「小心月台間隙」，即地鐵車門與月台之間的空隙）警示標語所述。

事實上，你的程式**現在**與**之後**部分之間的關係，正是非同步程式設計的核心所在。

當然，從一開始 JS 就有非同步程式設計（asynchronous programming）這種東西了，但多數 JS 開發人員從未真的仔細思考它們如何及為何突然出現在他們的程式之中，也沒有去探索應付它們的其他方式。**夠好就好**（*good enough*）的做法一直都是單純的回呼函式（callback function），即使到了今天，仍有許多人會堅稱 callbacks 就非常夠用了。

但為了因應身為一級程式語言（first-class programming language），在瀏覽器與伺服器和其間各種想像得到的裝置上運行而不斷拓展的需求，JS 在應用範圍與複雜度上也持續成長，管理非同步（asynchrony）處理的苦痛隨之加劇，我們需要更有能力且更容易推理（reason-able）的做法。


碁峰 www.gotop.com.tw

雖然所有的這些目前看起來相當抽象，我向你保證在本書後續的說明中，我會更完整且具體地處理這個主題。接下來的幾章，我們將會探索用於非同步 JavaScript（async JavaScript）程式設計的各種新興技巧。

但在那之前，我們得先更深入地了解什麼是非同步，以及它在 JS 中如何運作。

## 切分成塊的程式

你可能是在單一個 *.js* 檔案中編寫你的 JS 程式，不過你的程式幾乎總是由數個組塊（chunks）所構成，其中只有一個**現在**（*now*）就會執行，而剩下的**之後**（*later*）才會執行。這種 *chunk* 最常見的單位是 `function`（函式）。

JS 新手似乎都會遭遇的問題是，這個 *later* 並不一定是在 *now* 之後即刻發生。換句話說，無法**現在**就完成的任務，依照定義，都將會非同步地（asynchronously）完成，因此我們才不會有阻斷行為（blocking behavior），如你直覺上預期或想要的那樣。

請考慮：

```
// ajax(..) 是某個程式庫所給的任意 Ajax 函式
var data = ajax( "http://some.url.1" );

console.log( data );
// 糟了！`data` 通常不會有 Ajax 的結果
```

你可能有察覺到，標準的 Ajax 請求（requests）不會同步（synchronously）完成，這代表那個 `ajax(..)` 函式尚沒有任何值可以回傳，還無法指定給 `data` 變數。如果 `ajax(..)` 會阻斷（block）直到回應（response）傳回，那麼 `data = ..` 指定就運作無礙。

但那並非我們使用 Ajax 的方式。我們**現在**發出了一個非同步的 Ajax 請求，但在**之後**才會得到結果。

「等候」從 *now* 到 *later* 這段時間最簡單（但絕對不是唯一，甚至不一定是最好）的方式是使用一個函式，通常叫做**回呼函式**（*callback function*）：

```
// ajax(..) 是某個程式庫所給的任意 Ajax 函式
ajax( "http://some.url.1", function myCallbackFunction(data){
```



碁峰 www.gotop.com.tw

```
    console.log( data ); // 好耶，我得到一些 `data` 了！

} );
```

你也許曾經聽過我們能夠發出同步的 Ajax 請求，雖然技術上來說那是正確的，你永遠都不應該那樣做，不管在任何情況下都是如此，因為那會鎖住瀏覽器的 UI（按鈕、選單、捲軸等），無法再處理任何使用者互動。那是非常糟糕的主意，應該永遠被禁止。

在你提出反對意見之前，我只能先說不：想要避開麻煩的 callbacks 並不是允許阻斷式同步 Ajax 的正當理由。

舉例來說，考慮這段程式碼：

```
function now() {
    return 21;
}

function later() {
    answer = answer * 2;
    console.log( "Meaning of life:", answer );
}

var answer = now();

setTimeout( later, 1000 ); // 生命的意義：42
```

這個程式分成兩塊：**現在**（*now*）就會執行的東西，以及**之後**（*later*）才會執行的東西。我應該已經相當清楚標明了哪部分程式碼屬於哪一塊，但就讓我們超級明確表示：

現在：

```
function now() {
    return 21;
}

function later() { .. }

var answer = now();

setTimeout( later, 1000 );
```



碁峰 www.gotop.com.tw

之後：

```
answer = answer * 2;
console.log( "Meaning of life:", answer );
```

*now* 那一塊只要你的程式一跑就會即刻執行。但 `setTimeout(..)` 也會設定一個事件（逾時事件）在 *later* 發生，所以 `later()` 函式的內容會在之後的時間（1,000 毫秒後）執行。

只要你將一部分的程式碼包進一個 `function`，並指明它應該在回應某些事件（計時器、滑鼠點擊、Ajax 回應等）時才執行，你就是在建立你程式碼的一個 *later* 組塊，因而在你的程式中引入了非同步處理。

## 非同步的 Console

`console.*` 方法的運作方式並沒有規格（specification）或需求（requirements）來規範，它們並非 JavaScript 官方的一部分，而是由**宿主環境**（*hosting environment*，參閱本系列的**型別與文法**）加到 JS 之上的。

所以，不同的瀏覽器或 JS 環境高興怎麼做就怎麼做，而這有時可能會導致令人困惑的行為。

特別是，在某些瀏覽器中或某些條件之下，`console.log(..)` 實際上並不會即刻輸出所給的東西。之所以會發生這種情況，主要是因為在許多程式（不只是 JS）中，I/O 都是非常緩慢而且會阻斷執行的部分。因此，瀏覽器若在背景以非同步的方式處理 `console` I/O，效能可能會比較好（從網頁與 UI 的角度來看），而你甚至不會知道它們正在那樣做。

這可能**觀察得到**（*observable*）但並不常見的場合（不是從程式碼本身，而是從外部）是：

```
var a = {
    index: 1
};

// 之後
console.log( a ); // ??

// 更之後
a.index++;
```



碁峰 www.gotop.com.tw

我們一般會預期看到 a 物件在 console.log(..) 執行時立刻就被捕捉快照（snapshotted），印出像是 { index: 1 } 的東西，如此一來在下一個述句（statement）中，a.index++ 出現時，它所修改的是與 a 的輸出不同的東西，或只是緊接在其後的東西。

大多數情況下，前面的程式碼很可能會在你開發人員工具的主控台（console）中產生你所預期的物件表示值（object representation）。但也有可能在同樣的程式碼執行的情境中，瀏覽器覺得有必要將主控台 I/O 推延（defer）到背景去，在這種情況下，物件被表現到瀏覽器主控台之時，a.index++ 已經發生過了，所以顯示的是 { index: 2 }。

你很難確定到底在何種情況下 console I/O 會被推延，甚至難以判斷何種情況下觀察得到，不過至少你要知道 I/O 中有這種可能的非同步行為存在，以防你在除錯時，遇到物件應該是在 console.log(..) 之後被修改的，但你卻看到非預期結果的情況。

如果你遇到了這種少見的情況，最好的選擇是改用你 JS 除錯器的中斷點（breakpoints），而不仰賴 console 輸出。其次則是將目標物件序列化（serializing）為一個 string（例如使用 JSON.stringify(..)），強制為它捕捉一個「快照（snapshot）」。

## 事件迴圈

讓我們做個（或許有點驚人）的斷言：儘管顯然允許非同步的 JS 程式碼（就像我們剛才見過的 timeout），但直到最近（ES6）之前，JavaScript 本身實際上都未曾內建有任何直接的非同步概念。

什麼！這聽起來像是瘋狂的斷言，對吧？事實上，它相當正確。JS 引擎（engine）本身所做的，從未超出「被要求時，就在任何給定的時間點執行你程式的單一組塊」的範圍。

「被要求」，被誰要求？那是很重要的部分！



碁峰 www.gotop.com.tw

JS 引擎並不是獨立運作的，它會在一個**宿主環境**（*hosting environment*）內執行，對大多數開發人員來說，那就是典型的 web 瀏覽器。過去幾年間（但並不代表更之前沒有過），JS 透過像是 Node.js 之類的東西擴展到了瀏覽器以外的其他環境中，例如伺服器（servers）。事實上，近來 JavaScript 已經被內嵌到各式各樣的裝置內，從機器人到燈泡都有。

不過所有的這些環境都有一個共通的「主軸（thread[1]）」（無論好壞，這都是個不怎麼微妙的非同步笑話）：它們之中都有一個機制負責在每次 JS 引擎被調用時，處理你程式的多個組塊**分布在各個時間**的執行工作，這個機制就叫做**事件迴圈**（*event loop*）。

換句話說，JS 引擎本身並沒有時間的概念，而是一種視需要（on-demand）處理任意 JS 程式碼片段的執行環境。這是永遠都**排程了**（*scheduled*）「事件」（JS 程式碼的執行）的周邊環境。

所以，舉例來說，當你的 JS 程式發出了一個 Ajax 請求要從某個伺服器擷取一些資料，你會在一個函式（通常叫做一個 *callback*）中編寫好回應程式碼（response code），而 JS 引擎會告知宿主環境說：「嘿，我現在要暫停執行，不過只要你完成了那個網路請求，有拿回了一些資料後，請回頭呼叫這個函式」。

然後瀏覽器就會監聽從網路進來的回應，而當它有了可以提供給你的東西時，它就會排程（schedules）那個回呼函式以便執行，也就是將它新增到事件迴圈中。

所以到底事件迴圈是什麼呢？

讓我們先透過一些虛構的程式碼來解說它的概念：

```
// `eventLoop` 是行為像是一個佇列（queue）的陣列（array）
// （先進，先出）
var eventLoop = [ ];
var event;

// 持續執行直到「永遠」
while (true) {
    // 時鐘滴答的「滴（tick）」
    if (eventLoop.length > 0) {
        // 取得佇列中的下一個事件
        event = eventLoop.shift();
```

1 thread 在非同步程式設計中，也有「執行緒」或「線程」的意思，此處作者用到了這個雙關語。



碁峰 www.gotop.com.tw

```
            // 現在，執行下一個事件
            try {
                event();
            }
            catch (err) {
                reportError(err);
            }
        }
    }
```

這當然是非常簡化的虛擬程式碼，只是為了展示事件迴圈的概念，但它應該足以幫助你更加了解事件迴圈。

如你所見，有一個持續執行的迴圈，由上面的 `while` 迴圈表示，而該迴圈的每次迭代（iteration）都代表時鐘滴答計時的一次**滴**（*tick*）。對於每一個 tick，如果佇列上有一個事件在等候，它就會被取出並執行，這些事件就是你的函式回呼（function callbacks）。

要注意的一個重點是，`setTimeout(..)` 不會將你的 callback 放在事件迴圈佇列（event loop queue）上，它所做的是設定一個計時器（timer），當計時器的時間到了，環境就會將你的 callback 放到事件迴圈中，這樣的話未來的某個 tick 就會取出它來執行。

如果那時事件迴圈中已經有 20 個項目在排隊了怎麼辦？你的 callback 就得等候，它得排在其他項目後面，一般來說，並沒有搶先執行或跳到隊伍前面的路徑。這解釋了 `setTimeout(..)` 計時器為什麼可能無法在精確的時間點上觸發。你得到的保證只是（大略來說）你的 callback 不會在你所指定的時間**之前**觸發，它有可能在那時間點上觸發，也有可能在那之後觸發，取決於事件佇列的狀態。

所以，換句話說，你的程式通常會被拆成很多小組塊（chunks），在事件迴圈中一個接著一個觸發，而嚴格來說，並不直接與你程式有關的事件也可能在佇列中交錯（interleaving）出現。


碁峰 www.gotop.com.tw

我們提過「直到最近之前」，指的是 ES6 變更了管理事件迴圈佇列的位置。雖然這是一般不用深究的技術細節，但主要就是指 ES6 現在有確切規範事件迴圈如何運作，這表示技術上來說，它現在也在 JS 引擎的管控範圍內，而非只由宿主引擎（hosting environment）來掌控。這個變更的主要原因之一是 ES6 所引進的 Promises（承諾）功能（我們會在第 3 章討論），因為它們需要對事件迴圈佇列上排程的作業有直接且細部的控制（參照後面章節「合作」中關於 `setTimeout(..0)` 的討論）。

## 執行緒的平行處理

我們常會把「async（非同步）」和「parallel（平行）」這兩個術語混為一談，但實際上它們相當地不同。記得，async 是關於 *now*（**現在**）和 *later*（**之後**）之間的間隙（gap），而 parallel 指的是能夠同時（simultaneously）發生。

平行計算（parallel computing）最常見的工具是**行程**（*processes*）和**執行緒**（*threads*）。行程與執行緒都獨立運作，並可同時執行：在個別的處理器上，或甚至在不同的電腦上，不過多個執行緒能夠共用單一個行程的記憶體。

相較之下，事件迴圈會將其工作拆成任務（tasks），並循序（in serial）執行，不允許平行存取，也不允許對共用記憶體的變更。平行處理（parallelism）和序列處理（serialism）可能會以「個別執行緒中彼此合作的事件迴圈」的形式並存。

平行執行緒的交錯執行和非同步事件的交錯發生是在非常不同的層次上運作的。

舉例來說：

```
function later() {
    answer = answer * 2;
    console.log( "Meaning of life:", answer );
}
```



碁峰 www.gotop.com.tw

雖然 later() 的整個內容可被視為事件迴圈佇列上的單一個條目（entry），但考慮這段程式碼所運行的執行緒時，實際上可能會有十幾個不同的低階作業發生。例如 answer = answer * 2 得先載入 answer 目前的值，再將 2 放到某個地方，接著執行乘法運算，然後取回結果並將之儲存到 answer 中。

在單執行緒（single-threaded）的環境中，執行緒佇列中的項目是否為低階作業並非真的很重要，因為沒有其他東西可以中斷該執行緒的執行，但如果你所在的是平行系統，而同一個程式中有兩個不同的執行緒在進行作業，你就很有可能會遇上非預期的行為。

請考慮：

```
var a = 20;

function foo() {
    a = a + 1;
}

function bar() {
    a = a * 2;
}

// ajax(..) 是某個程式庫所給的任意 Ajax 函式
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

在 JavaScript 的單執行緒行為中，如果 foo() 在 bar() 之前執行，那結果就是 a 有 42 的值，但若 bar() 是在 foo() 之前執行，那 a 中的結果就會是 41。

如果共用相同資料的 JS 事件平行運作，那麼問題就會變得微妙許多。請考慮這兩串虛擬程式碼任務，如果它們是能夠分別執行 foo() 和 bar() 中程式碼的執行緒，而它們又在完全相同的時間點執行，那會發生什麼事？

Thread 1（X 與 Y 是暫存用的記憶體位置）：

```
foo():
  a. load value of `a` in `X`
  b. store `1` in `Y`
  c. add `X` and `Y`, store result in `X`
  d. store value of `X` in `a`
```


碁峰 www.gotop.com.tw

Thread 2（X 與 Y 是暫存用的記憶體位置）：

```
bar():
  a. load value of `a` in `X`
  b. store `2` in `Y`
  c. multiply `X` and `Y`, store result in `X`
  d. store value of `X` in `a`
```

現在，讓我們假設這兩個執行緒真的平行運作。你大概能夠看出問題所在了，對吧？它們在其暫存步驟中用到了共用的記憶體位置 X 與 Y。

如果那些步驟發生的順序如下，那麼最後的結果會是什麼呢？

```
1a  (load value of `a` in `X`   ==> `20`)
2a  (load value of `a` in `X`   ==> `20`)
1b  (store `1` in `Y`   ==> `1`)
2b  (store `2` in `Y`   ==> `2`)
1c  (add `X` and `Y`, store result in `X`   ==> `22`)
1d  (store value of `X` in `a`   ==> `22`)
2c  (multiply `X` and `Y`, store result in `X`   ==> `44`)
2d  (store value of `X` in `a`   ==> `44`)
```

a 中的結果會是 44。那如果順序是這樣呢？

```
1a  (load value of `a` in `X`   ==> `20`)
2a  (load value of `a` in `X`   ==> `20`)
2b  (store `2` in `Y`   ==> `2`)
1b  (store `1` in `Y`   ==> `1`)
2c  (multiply `X` and `Y`, store result in `X`   ==> `20`)
1c  (add `X` and `Y`, store result in `X`   ==> `21`)
1d  (store value of `X` in `a`   ==> `21`)
2d  (store value of `X` in `a`   ==> `21`)
```

a 中的結果會是 21。

所以，帶有執行緒的程式設計非常棘手，因為如果你不採取特殊的步驟來防止這種中斷或交錯的情形發生，你可能會得到非常出乎意料、非確定性的（nondeterministic）行為，而那經常會讓你頭痛不已。

JavaScript 從來都不會跨執行緒共用資料，這表示非確定性不是需要考量的問題，但這並不代表 JS 永遠都是確定性的（deterministic）。還記得前面一點，我們提過 foo() 與 bar() 的相對順序會產生不同的結果（41 或 42）嗎？



碁峰 www.gotop.com.tw

目前或許還看不太出來，但並非所有的非確定性（nondeterminism）都是不好的，有時那至關緊要，而有時那是刻意為之的。我們會在本章與接下來的幾章看到更多的範例。

## 執行至完成

因為 JavaScript 是單執行緒的，foo()（及 bar()）內的程式碼是不可部分完成的（atomic），這表示一旦 foo() 開始執行，它所有的程式碼都要執行完成，bar() 中的程式碼才能執行，反之亦然。這種行為叫做 *run-to-completion*（執行至完成）。

事實上，這種 run-to-completion 的語意（semantics）要在 foo() 與 bar() 中有更多的程式碼之時才會更為明顯，例如：

```
var a = 1;
var b = 2;

function foo() {
    a++;
    b = b * a;
    a = b + 3;
}

function bar() {
    b--;
    a = 8 + b;
    b = a * 2;
}

// ajax(..) 是某個程式庫所給的任意 Ajax 函式
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

因為 foo() 不會被 bar() 中斷，而 bar() 也不會被 foo() 中斷，這個程式只會有兩個可能的結果，取決於何者先開始執行，但如果有多執行緒，而 foo() 與 bar() 中的個別述句（statements）可以交錯執行，那麼可能結果的數目就會大幅增加！

Chunk 1 是同步的（**現在**就發生），但 Chunk 2 與 Chunk 3 都是非同步的（**之後**才發生），這表示它們的執行會有時間上的間隔。



碁峰 www.gotop.com.tw

Chunk 1：

```
var a = 1;
var b = 2;
```

Chunk 2（foo()）：

```
a++;
b = b * a;
a = b + 3;
```

Chunk 3（bar()）：

```
b--;
a = 8 + b;
b = a * 2;
```

Chunks 2 與 Chunk 3 任一者都可能先發生，所以此程式有兩種可能的結果（outcomes），如這裡所示：

Outcome 1:：

```
var a = 1;
var b = 2;

// foo()
a++;
b = b * a;
a = b + 3;

// bar()
b--;
a = 8 + b;
b = a * 2;

a; // 11
b; // 22
```

Outcome 2：

```
var a = 1;
var b = 2;

// bar()
b--;
a = 8 + b;
b = a * 2;

// foo()
```



碁峰 www.gotop.com.tw

```
a++;
b = b * a;
a = b + 3;

a; // 183
b; // 180
```

同樣的程式碼有兩種可能的結果就代表我們仍然有非確定性！但這是在函式（事件）的順序層級（ordering level），而非使用執行緒時的述句順序層級（或是實際上的運算式作業順序層級）。換句話說，這比使用執行緒的情況更有確定性。

套用到 JavaScript 的行為時，這種函式順序（function-ordering）的非確定性常被稱為**競態狀況**（*race condition*），因為 `foo()` 與 `bar()` 彼此競爭看誰會先執行。具體而言，這之所以是競態狀況，是因為你無法可靠地預測 `a` 與 `b` 的結果會是什麼。

如果 JS 中有個功能沒有這種 run-to-completion 的行為，我們就會有更多可能的結果，對吧？ ES6 確實引進了這樣的東西（參閱第 4 章），但現在先別擔心，我們之後就會討論到！

## 共時

讓我們想像有個網站會顯示一連串的狀態更新（就像社群網路的動態消息），它會隨著使用者往下捲動漸進式地載入更多內容。要使這種功能正確運作，（至少）有兩個不同的「行程（processes）」必須**同時**（*simultaneously*）執行（也就是在同一小段時間窗口內，但並不一定在同一瞬間）。

這裡的「行程（process）」之所以要放在引號內，是因為它們不是真正作業系統層級的行程，有別於電腦科學中的用法。它們是虛擬行程（virtual processes）或任務，表示一系列邏輯上相連且循序的作業（operations）。我們使用「行程」而非「任務」，是因為從術語上來說，它比較符合我們所探討的概念之定義。



碁峰 www.gotop.com.tw

第一個「行程」負責回應使用者向下捲動頁面所觸發的 `onscroll` 事件（製作 Ajax 請求來要求新內容）。第二個「行程」負責接收 Ajax 回應（以重新描繪頁面內容）。

很明顯，如果使用者捲動得夠快，你可能會在第一個回應傳回並被處理的過程中，看到兩個或更多個 `onscroll` 事件，所以會有 `onscroll` 事件與 Ajax 回應事件彼此交錯快速觸發。

共時（concurrency）是當兩個或更多個「行程」在同一段時間內同時執行，不管構成它們的個別作業是否**平行**（*in parallel*，個別處理器或核心上的同一瞬間）發生。因此你可以把共時想成是「行程」層級（或任務層級）的平行處理，相對於作業層級的平行處理（operation-level parallelism，個別處理器的執行緒）。

共時也引入了這些「行程」彼此互動的選擇性概念，我們之後會回到這個主題上。

在一個給定的時間窗口（window of time，使用者捲動的數秒）內，讓我們將每個獨立的「行程」視覺化為一系列的事件或作業：

「Process」1（`onscroll` 事件）：

```
onscroll, request 1
onscroll, request 2
onscroll, request 3
onscroll, request 4
onscroll, request 5
onscroll, request 6
onscroll, request 7
```

「Process」2（Ajax 回應事件）：

```
response 1
response 2
response 3
response 4
response 5
response 6
response 7
```



碁峰 www.gotop.com.tw

相當有可能的情況是，一個 onscroll 事件與一個 Ajax 回應事件剛好在同一瞬間準備就緒得以處理。舉例來說，讓我們在一個時間軸視覺化這些事件：

```
onscroll, request 1
onscroll, request 2          response 1
onscroll, request 3          response 2
response 3
onscroll, request 4
onscroll, request 5
onscroll, request 6          response 4
onscroll, request 7
response 6
response 5
response 7
```

但是，回到本章前面我們提過的事件迴圈概念，JS 一次只能處理一個事件，所以要不是 onscroll, request 2 先發生，就是 response 1 會先發生，但它們無法在同一瞬間發生。就像在學校食堂的小朋友，不論他們是從門外的哪群學生中竄出，最終都還是得排到某個隊伍中，才能拿到他們的午餐！

讓我們將所有這些交錯的事件視覺化到事件迴圈佇列（event loop queue）之上：

```
onscroll, request 1   <--- Process 1 開始
onscroll, request 2
response 1            <--- Process 2 開始
onscroll, request 3
response 2
response 3
onscroll, request 4
onscroll, request 5
onscroll, request 6
response 4
onscroll, request 7   <--- Process 1 完成
response 6
response 5
response 7            <--- Process 2 完成
```

「Process」1 與「Process」2 共時地執行（任務層級的平行），但它們的個別事件則是在事件迴圈佇列上循序地（sequentially）執行。

順道一提，有注意到 response 6 與 response 5 傳回的順序出乎意料嗎？


碁峰 www.gotop.com.tw